MECHANISM BY WHICH PLATFORM INDEPENDENT SOFTWARE MAY BIND TO AND ACCESS PLATFORM DEPENDENT SOFTWARE

INVENTORS:

Gregory L. Slaughter

Thomas E. Saulpaugh

Bernard A. Traversat

BACKGROUND OF THE INVENTION

The Field of the Invention

The present invention relates to the field of computing, and more particularly, to software, methods and systems that facilitate platform independent software binding to and accessing platform dependent software.

Description of Relevant Art

In recent years significant efforts have been made to facilitate the creation of platform independent software. This is, software that can execute on multiple different platforms (e.g. different types of computer system), without requiring the software program to be rewritten or customized to operate on specific platforms. By way of example, the Java programming language is designed to facilitate the creation of Java software objects that are platform independent. However, in certain situations, it is desirable for a particular software object to make use of specific platform dependent code to access specific hardware functionality. One such class of software objects is device drivers.

A device driver is software used to control hardware coupled with the computer system. Usually, the peripheral device functions to provide data input and/or output ("I/O") to the computer system. Examples of peripheral devices include keyboards, printers, scanners, network interface and graphics cards, modems, and monitors. In general, device drivers process data being to be sent to or retrieved from the peripheral device by the computer system so that

the data is transferred in a format suitable for processing by the peripheral device or computer system.

The intimate association between the device driver and the hardware and software of both the peripheral device and the computer system to which the device is couple has required that device drivers be written in a highly platform-dependent manner. For example, device drivers generally must obtain memory resources when called to perform their function. Typically, this requires the allocation of memory resources that must be described by the driver. The driver must therefor have specific knowledge about the platform in order to make such a request. Thus, the same peripheral device, e.g., a printer, will require different version of device (printer) driver software for each platform.

The platform dependence of driver software thus increases the costs of developing platforms and peripherals, as manufactures of peripherals and computer operating systems must provide new versions and updates of driver software for new peripherals, new software platforms, and new operating system releases. Platform-dependent driver technology also increases the cost of maintaining computer systems, especially diverse computer systems deployed over networks, as system managers must obtain and install new and updated device drivers to enable user access to peripheral devices.

One approach to solving the problem of running platform independent software objects with platform dependent software (such as drivers) is to provide software objects that include one part that is platform independent and another part that is platform dependent. However, this "mixed" software approach is generally undesirable since such mixed objects require the platform dependent part to include different versions of the platform dependent code for each platform that the service will ultimately be loaded upon. Maintaining so many different versions in the same software object is both unwieldy and wasteful.

Therefore, it would be advantageous to provide a mechanism that provides for platform-independent software objects to efficiently bind to and access platform dependent software.

SUMMARY OF THE INVENTION

Methods and apparatus that permit platform independent software objects to access platform dependent methods are disclosed. According to one aspect of the present invention, a software object is described. The software object includes a platform dependent method and a wrapper arranged to call the platform dependent method. In one embodiment, a platform independent object accesses the platform dependent method by calling the wrapper which then calls the platform dependent method.

In some embodiments, a secure accessing arrangement is provided wherein the platform independent object requests the encapsulation object containing the platform dependent method from the system manager. A business card corresponding to the requesting object is retrieved by the system manager after which the system manager retrieves an encapsulation object pointer used to identify the encapsulation object from the business card. The identified encapsulation object is then instantiated after which the wrapper corresponding to the platform dependent method is called by the platform independent object. The wrapper then calls the platform dependent method.

According to still another aspect of the invention, a computer implemented method for transforming a mixed software object containing platform independent code and platform dependent code into a pure software object having platform independent code only is disclosed. The method includes writing an encapsulation object containing all the platform dependent code. The native encapsulation object includes a wrapper used to invoke the platform dependent code. Next, all the platform dependent code is removed from the mixed software object and all calls to the platform dependent code in the mixed software object are replaced with calls to the wrapper included in the native encapsulation object.

These and other advantages of the present invention will become apparent upon reading the following detailed descriptions and studying the various figures of the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a native encapsulation object (NEO) in accordance with an embodiment of the invention.

Figure 2 is a flowchart detailing a process for a platform independent object to access platform dependent software in accordance with an embodiment of the invention.

Figure 3 is a flowchart detailing a process for a platform independent device driver binding to and accessing platform dependent software in accordance with the process detailed in Figure 2.

Figure 4 shows a flowchart detailing a process for transforming a mixed software object having both platform independent and platform dependent code to a purely platform independent software object in accordance with an embodiment of the invention.

Figure 5 is a schematic illustration of a computer system, or "platform", in accordance with the present invention.

Figure 6 is a diagram illustrating an object-oriented operating system in accordance with an embodiment the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The present invention includes software, methods, and apparatus that are arranged to permit platform-independent objects to access platform dependent software. In one described embodiment of the present invention, an encapsulation object that includes a wrapper arranged to call an associated platform independent method is described. In the described embodiment, substantially the only operation performed by the wrapper is to act as an intermediary between a platform independent object and the platform dependent method to facilitate calling the platform dependent method from the platform independent service.

Figure 1 illustrates a native encapsulation object 100 in accordance with an embodiment of the invention. The native encapsulation object 100 includes wrappers 102 and corresponding platform dependent methods (also referred to as native methods) 104. In one embodiment of the invention, each of the wrappers 102 is arranged to encapsulate (or call) an associated one of the platform dependent methods 104. In this way, a platform independent object, such as for example, a device driver, can access platform dependent software (represented by the native methods) using a platform independent protocol by first calling the wrappers 102 associated with the platform independent software. The wrapper, in turn, calls (or invokes) the appropriate platform independent software, which is then accessible to the platform independent object.

In one particular embodiment, some of the wrappers 102 take the form of Java wrapper 102a through 102n that are well known to those skilled in the art. By way of example, the Java wrapper 102a is associated with a platform dependent method 104a included in the methods 104. A platform independent object 106, such as a device driver, can preserve its platform independence by first calling the Java wrapper 102a instead of directly calling the native method 104a. The Java wrapper 102a responds by directly calling the platform dependent method 104a. The method 104a is then accessible to the platform independent object 106. Since the object 106 has only called the Java wrapper 102a (which is by definition platform

independent) the object 106 is able to access platform dependent software represented by the native methods 104 and yet maintain its platform independence. It should be noted that the use of a Java wrapper was exemplary and should not be construed as limiting the scope or intent of the invention since any programming language can be used.

Figure 2 is a flowchart detailing a process 200 for allocating a native encapsulation object containing platform dependent software to be accessed by a platform independent object in a computer system in accordance with an embodiment of the invention. It should be noted that the process 200 is applicable to any platform independent software including, but not limited to, for example, device drivers.

The process 200 begins at 202 by the platform independent object calling a system facility. In one embodiment, the system facility is, in the case of Java, a bus manager. At 204, the system facility looks up and retrieves a business card associated with the requesting platform independent object. In the described embodiment, all platform independent objects have a corresponding business card that is instantiated at system startup. The business card includes configuration data that is used to identify the native encapsulation object containing the platform dependent software. At 206, the native encapsulation object containing the platform dependent software is identified by the business card. At 208, the identified native encapsulation object is instantiated and at 210 the identified native encapsulation object is returned to the requesting object.

Figure 3 is a flowchart details a process 300 whereby a platform independent device driver accesses platform dependent software. It should be noted that the process 300 is a specific implementation of the process 200. Conceptually, the process 300 can be divided into an allocation portion represented by blocks 302 through 308 and an accessing portion 310 through 312. It should be realized, however, that the allocating and accessing portions are described for clarity purposes only.

The allocation portion of the process 300 begins at 302 by the platform independent device driver requesting a native encapsulation object that contains the appropriate platform dependent native methods from a bus manager connected to the device driver. The bus manager responds by returning a business card associated with the requesting device driver at 304. It should be noted, that each device driver has associated with it a business card that is, in one embodiment, populated by the system administrator at system set up with configuration data fields specific to that device driver. Such configuration data includes, in one embodiment, a NEO name useful in identifying a particular native encapsulation object such as the native encapsulation object containing the native methods to be accessed by the device driver. In a particular embodiment, the native encapsulation object pointer takes the form of a string of characters referred to as a native encapsulation object name. The native encapsulation object pointer corresponding to the native encapsulation object containing the native methods is then returned at 306. The native encapsulation object containing the native methods and identified by the returned native encapsulation object pointer is then instantiated and returned to the device driver at 308. At this point the allocation portion is substantially complete and the accessing portion of the process 300 begins by the platform independent device driver accessing the native methods by calling the wrappers included in the native encapsulation object at 310. The wrapper responds by calling the corresponding native methods at 312.

In some cases, it is desirable to implement the above-described calls from a device driver to a bus manager (e.g., for other system services such as connecting, disconnecting, or interrogating interrupts) in a secure way. For example, it may be desirable to prevent a "rogue" device driver from identifying itself to a bus manager as another, legitimate driver. This could be useful to improve system robustness, e.g., by allowing the system to identify and isolate corrupted device drivers and thus avoid system crashes. Such a security mechanism could also be useful to prevent "spoofing" device drivers from erasing, altering, and/or copying sensitive data from the system surreptitiously. By way of example, a malicious spoofing device driver could masquerade as a printer driver and copy the data to be printed to a remote location.

One embodiment of such a security mechanism is described in detail in patent application entitled, "SECURITY FOR PLATFORM-INDEPENDENT DEVICE DRIVERS", (serial no. 09/106,912) by Slaughter, et al filed June 29, 1998 which is incorporated in its entirety. In this way, a particular native encapsulation object whose native code performs critical functionality that affects the system hardware can be managed in a secure manner.

In another embodiment of the invention, mechanisms whereby a software object having both platform independent and platform dependent code can be transformed into two portions. One portion being platform independent and thereby loadable on any platform. The other portion being platform dependent and, as such, can be added to any platform on which the loadable platform independent portion may run.

Figure 4 shows a flowchart of a possible process 400 for transforming a mixed software object into platform independent and platform dependent portions in accordance with an embodiment of the invention. The process 400 begins at 402 by writing a native encapsulation object that contains all the platform dependent code contained in the mixed software object. Once the native encapsulation object has been written, the platform dependent code (native methods) is removed from the mixed software object at 404. Next, all calls to the native methods of the mixed software object are replaced by calls to the wrappers contained in the native encapsulation object at 406. In this way, the mixed software object has been transformed to a platform independent software.

Figure 5 shows an example of a computer system 500 suitable for implementation of the present invention. Computer system 500 includes a central processing unit ("CPU") 502, such as, for example, a Sun Microsystems SPARC, Motorola PowerPC, or Intel Pentium processor. CPU 502 is coupled in turn with memory 504. Memory 504 can include any type of memory device used in a computer system, including random access memory ("RAM") and read-only memory ("ROM"). CPU 502 is also coupled with a bus 506, such as a PCI bus, or an S bus. A variety of input devices 508 and 510, and output devices 512 are also coupled with bus 506. Examples of such input and output devices include, but are not limited to, printers, monitors, modems, and/or network/telephone connections. Typically each of these devices has an

associated driver as will be described in further detail below. Thus, for example, input device 508 could be a network interface card that connects computer system 500 to a local area network ("LAN"), input device 510 could be a keyboard, and output device 512 could be a monitor. CPU 502 can further be coupled to mass storage 514, such as a hard disk or CDROM drive, and DMA controller 516. The connection between CPU 502 and mass storage 514 can be over a second bus 518, such as a small computer system interface ("SCSI") bus.

Although computer system 500 has been illustrated using a certain configuration of hardware elements, it will be appreciated that the invention is not limited to operation on such configurations alone. Thus, computer system 500 is representative of computer systems in general terms and includes a wide variety of computing devices including, but not limited to, personal computers, mainframe or minicomputers, and "smart" systems such as set-top boxes used to access high definition television, cable, or satellite transmission as well as cellular telephones. Still more examples of computer systems suitable for use with the present invention will be apparent to those of skill in the computer science and electronics arts.

Figure 6 illustrates at 600 an example of software, such as stored in memory 504 and/or running on CPU 502, arranged in a series of layers. The two upper-most layers 602 include software that is platform-independent, i.e., software that is not specially configured to run on a particular computer system but that can be run on any of several computer systems. The remaining lower layers 604 are platform-dependent software, i.e., software that must be written especially for a particular computer system.

Layers 602 include an applications layer 606. Layer 606 includes software applications that are commonly run by computer users. Such software includes word processors, graphics programs, database programs, and desktop publishing programs. These applications run in conjunction with runtime system 608. Runtime system 608 includes, in one embodiment, a Java Virtual Machine ("JVM") 610 that receives instructions in the form of machine-independent bytecodes produced by the application running in applications layer 606 and interprets the instructions by converting and executing them. Other types of virtual machine

can be used with the present invention, however, as will be apparent to those of skill in the computer science and electronics arts.

Runtime system 608 further includes a set of additional functions 612 that support facilities such as I/O, network operations, graphics, printing, and the like. Also included with runtime system 608 is device interface 614 that supports the operation of buses 506 and 518, and devices 508, 510, and 512. Device interface 614 includes device drivers 616, which are object-oriented programs written to support the various devices coupled with computer system 500 such as devices 508, 510, and 512. In a preferred embodiment of the invention, each of the device drivers 616 has associated with it a business card 617. The business card 617 includes configuration data specific to the associated device driver 616.

Runtime system 608 further includes device managers 618, platform-independent memory 620, and platform-independent bus managers 622 that support buses 506 and 518. Thus, it will be appreciated that device drivers 616 and bus managers 622 can be used with any platform, including platforms yet to be developed. Other various managers and components (not shown) are typically provided with device interface 614 and will be known among those skilled in the computer science and electronics arts.

Platform-dependent layers 604 include platform interface 624 that contains DMA classes 626, bus managers 628, and memory classes 630 in addition to other facilities that support runtime system 608. Additional functions are also included in platform interface 624 including interrupt classes (not shown) to support computer system 500. In one embodiment, these classes, managers, and functions are written in the Java programming language.

OS native methods 632 includes DMA native methods 634 and memory native methods 636 that are written in a language specific to CPU 502 (and thus are "native"). These methods interface with microkernel 638. Finally, at the lowest layer is boot interface 640 that supports the loading and initialization of software into memory when computer system 500 is started.

Thus, the described inventions provide methods, software, and systems for platform-independent system services. Such system services include connecting, disconnecting, or interrogating interrupts, as well as device drivers. Such device drivers can be run in a secure fashion as described above and allow greater flexibility and reduced costs of construction and maintenance. Although certain embodiments and examples have been used to describe the present invention, it will be apparent to those having skill in the art that various changes can be made to those embodiment and/or examples without departing from the scope or spirit of the present invention. For example, it will be appreciated from the foregoing that many object oriented operating systems can be used to implement the methods, software, and systems described herein using only modifications that will be apparent to those of skill in the computer science arts.

What is claimed is: